Lecture 8

Dr Richard Reilly
Dept. of Electronic & Electrical Engineering

Room 153,
Engineering Building

Address Space

- **The address space** is the maximum amount of memory that a processor can address.
  - Some processors use a multi-level addressing scheme, with main memory divided into **segments** or **pages** and some or all instructions mapping into the current segment(s) or page(s).

Addressing Modes

- One of the most traditional ways of describing processor architectures is in terms of the number of addresses contained in each instruction.
- The most basic way of addressing an operand is to use direct addressing.
  - address of the operand is included in the instruction,

```
e.g.     MOVE    $1000,ACC          ; ACC = M[$1000]
```

- Modern CPUs provide a rich set of addressing modes.
  - The Intel 386 family has 11 different addressing modes.

Addressing Modes

- **Address field(s) in a typical instruction format are limited.**
  - Better to be able to reference a large range of locations in memory.

  ⇒ to achieve this objective, a variety of address methods/modes can be employed.
- Address methods/modes require some trade-off between the number of memory references and the complexity of address calculation.

Advantages of Addressing Modes

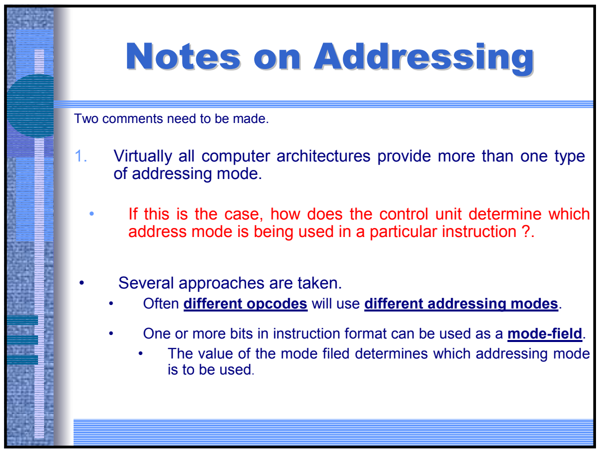
3 reasons

1. **Efficiency**
   The address section of the instruction word can be shortened and hence the execution can be faster.
2. **Flexibility**
   Certain addressing modes are more suited to some tasks than direct addressing.
   - working with arrays.
3. **Re-locatable Code**
   In a multi-tasking environment programs are loaded into memory by the operating system, the run time location of the code may be different each time the program is run.

Addressing Modes

The most commonly used addressing modes

- **Direct**
- **Immediate**
- **Register Indirect**
- **Register Indirect with Pre/Post - Inc/Dec**
- **Register Indexed**
- **Register Displaced**
- **Direct Register**



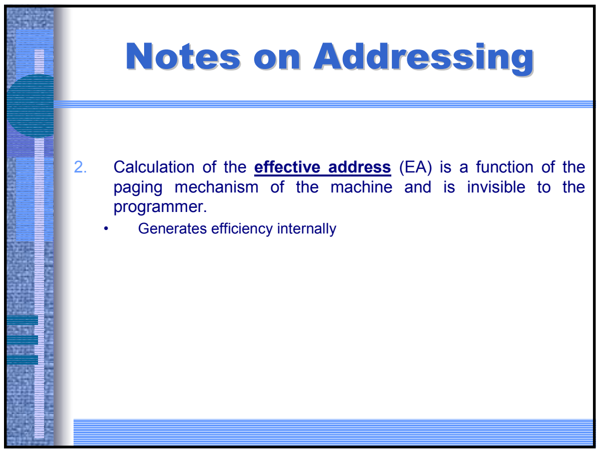
Notes on Addressing

Two comments need to be made.

1. Virtually all computer architectures provide more than one type of addressing mode.

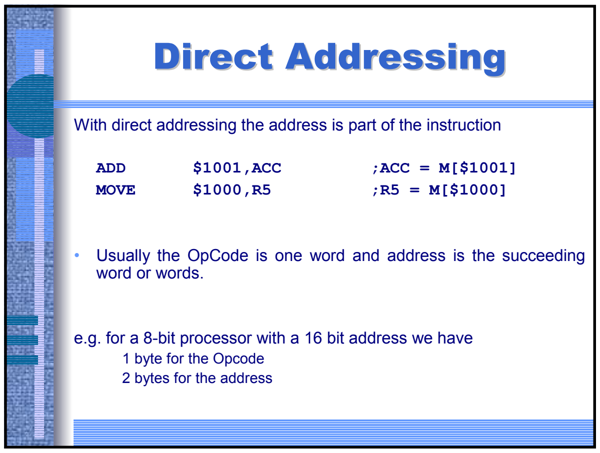
   - If this is the case, how does the control unit determine which address mode is being used in a particular instruction ?.

- Several approaches are taken.
  - Often **different opcodes** will use **different addressing modes**.
  - One or more bits in instruction format can be used as a **mode-field**.
    - The value of the mode filed determines which addressing mode is to be used.



Notes on Addressing

2. Calculation of the **effective address** (EA) is a function of the paging mechanism of the machine and is invisible to the programmer.
   - Generates efficiency internally



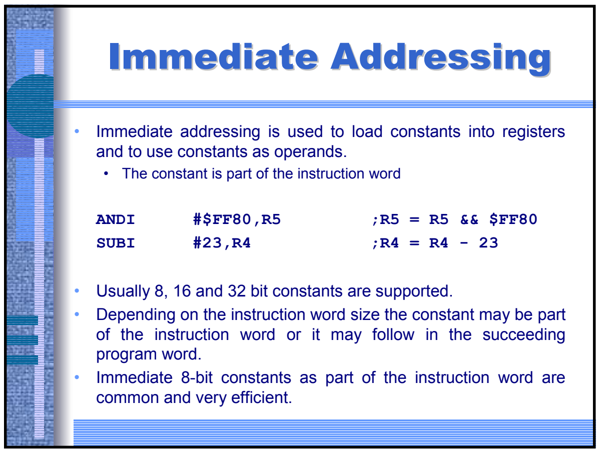
Direct Addressing

With direct addressing the address is part of the instruction

```
ADD      $1001,ACC        ;ACC = M[$1001]
MOVE     $1000,R5         ;R5 = M[$1000]
```

- Usually the OpCode is one word and address is the succeeding word or words.

e.g. for a 8-bit processor with a 16 bit address we have
1 byte for the Opcode
2 bytes for the address

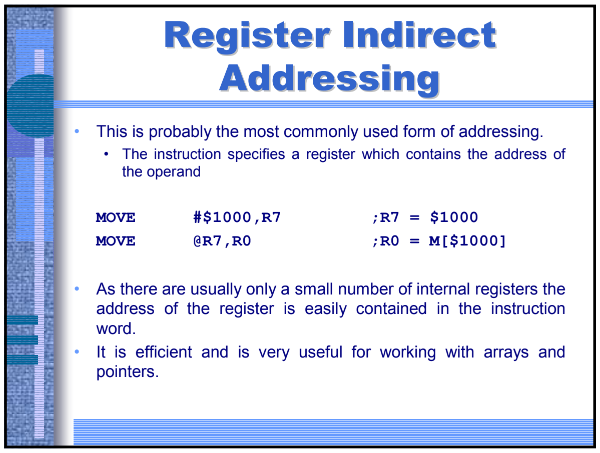


Immediate Addressing

- Immediate addressing is used to load constants into registers and to use constants as operands.
  - The constant is part of the instruction word

```
ANDI     #$FF80,R5        ;R5 = R5 && $FF80
SUBI     #23,R4           ;R4 = R4 - 23
```

- Usually 8, 16 and 32 bit constants are supported.
- Depending on the instruction word size the constant may be part of the instruction word or it may follow in the succeeding program word.
- Immediate 8-bit constants as part of the instruction word are common and very efficient.

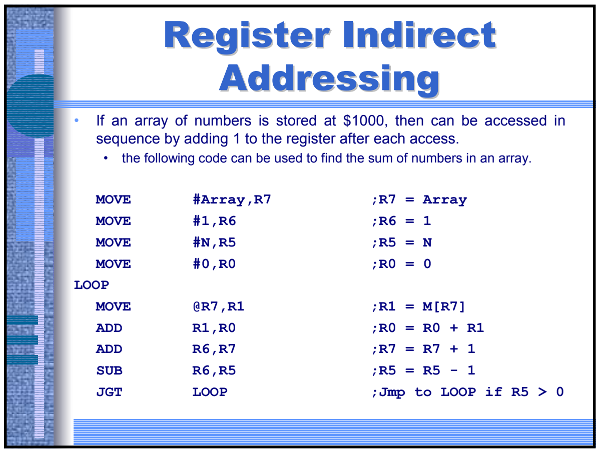


Register Indirect Addressing

- This is probably the most commonly used form of addressing.
  - The instruction specifies a register which contains the address of the operand

```
MOVE     #$1000,R7        ;R7 = $1000
MOVE     @R7,R0           ;R0 = M[$1000]
```

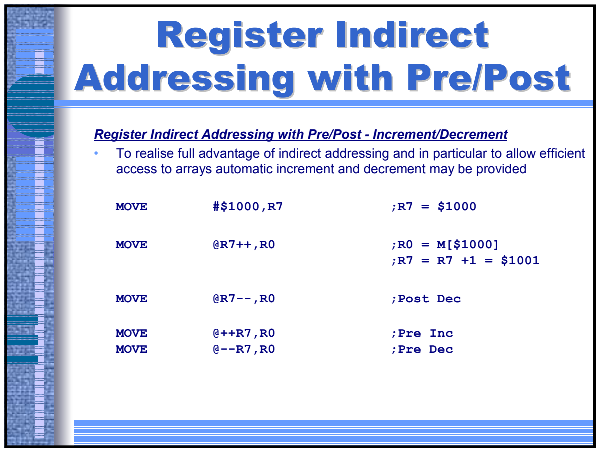
- As there are usually only a small number of internal registers the address of the register is easily contained in the instruction word.
- It is efficient and is very useful for working with arrays and pointers.



Register Indirect Addressing

- If an array of numbers is stored at $1000, then can be accessed in sequence by adding 1 to the register after each access.
  - the following code can be used to find the sum of numbers in an array.

```
  MOVE     #Array,R7        ;R7 = Array
  MOVE     #1,R6            ;R6 = 1
  MOVE     #N,R5            ;R5 = N
  MOVE     #0,R0            ;R0 = 0
LOOP
  MOVE     @R7,R1           ;R1 = M[R7]
  ADD      R1,R0            ;R0 = R0 + R1
  ADD      R6,R7            ;R7 = R7 + 1
  SUB      R6,R5            ;R5 = R5 - 1
  JGT      LOOP             ;Jmp to LOOP if R5 > 0
```

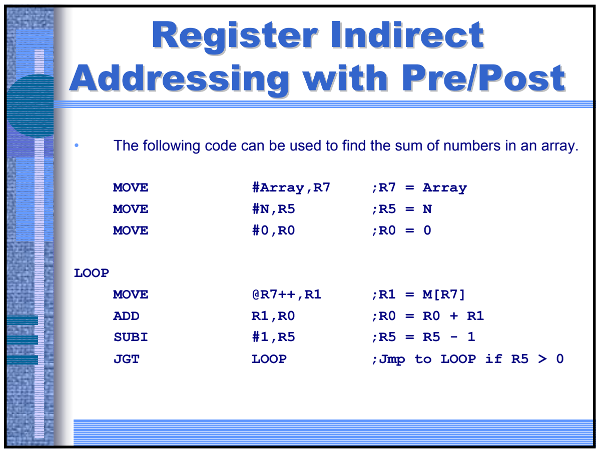


Register Indirect Addressing with Pre/Post

Register Indirect Addressing with Pre/Post - Increment/Decrement

- To realise full advantage of indirect addressing and in particular to allow efficient access to arrays automatic increment and decrement may be provided

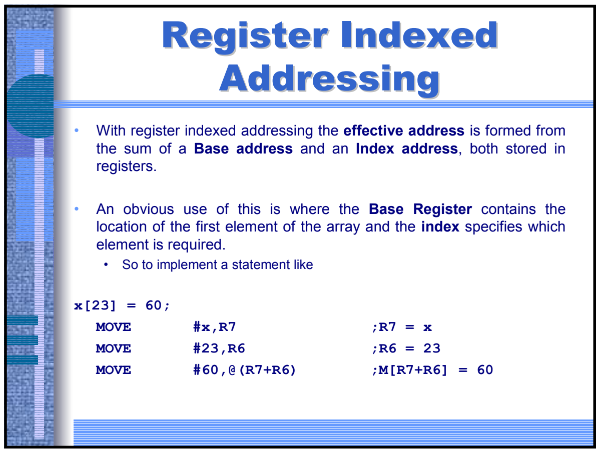
```
MOVE        #$1000,R7             ;R7 = $1000

MOVE        @R7++,R0              ;R0 = M[$1000]
                                  ;R7 = R7 +1 = $1001

MOVE        @R7--,R0              ;Post Dec

MOVE        @++R7,R0              ;Pre Inc
MOVE        @--R7,R0              ;Pre Dec
```



Register Indirect Addressing with Pre/Post

- The following code can be used to find the sum of numbers in an array.

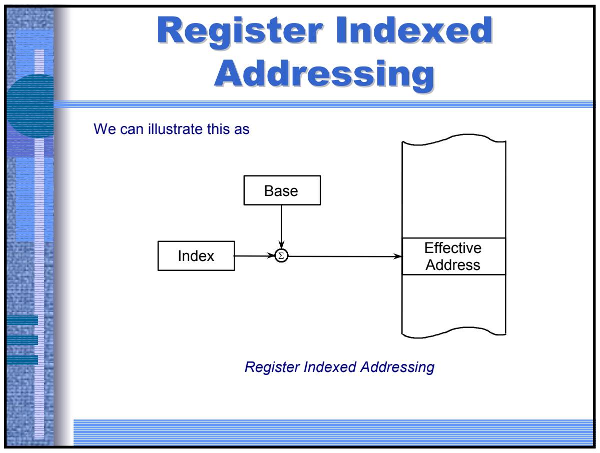
```
     MOVE            #Array,R7     ;R7 = Array
     MOVE            #N,R5         ;R5 = N
     MOVE            #0,R0         ;R0 = 0

LOOP
     MOVE            @R7++,R1      ;R1 = M[R7]
     ADD             R1,R0         ;R0 = R0 + R1
     SUBI            #1,R5         ;R5 = R5 - 1
     JGT             LOOP          ;Jmp to LOOP if R5 > 0
```



Register Indexed Addressing

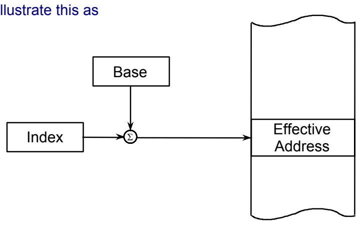
- With register indexed addressing the **effective address** is formed from the sum of a **Base address** and an **Index address**, both stored in registers.
- An obvious use of this is where the **Base Register** contains the location of the first element of the array and the **index** specifies which element is required.
  - So to implement a statement like

```
x[23] = 60;
   MOVE        #x,R7                 ;R7 = x
   MOVE        #23,R6                ;R6 = 23
   MOVE        #60,@(R7+R6)          ;M[R7+R6] = 60
```

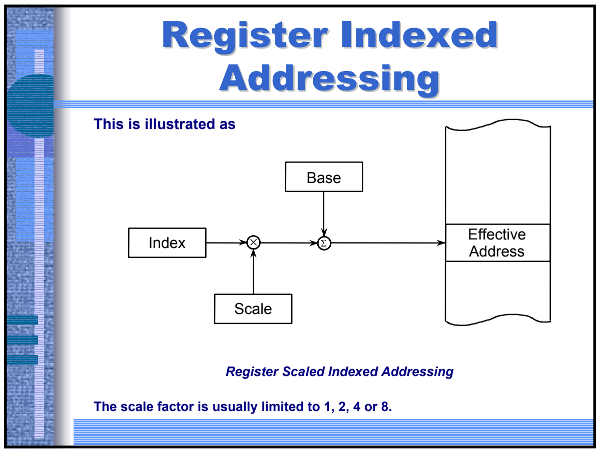


Register Indexed Addressing

We can illustrate this as

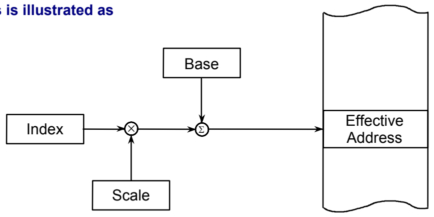


Register Indexed Addressing



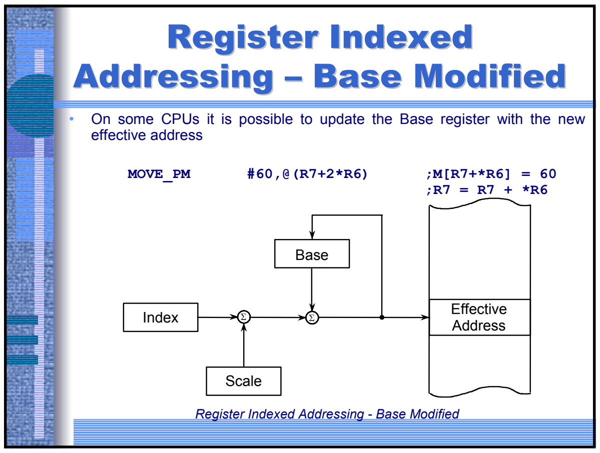
Register Indexed Addressing

This is illustrated as



Register Scaled Indexed Addressing

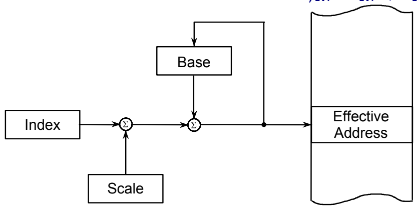
The scale factor is usually limited to 1, 2, 4 or 8.



Register Indexed Addressing – Base Modified

- On some CPUs it is possible to update the Base register with the new effective address

```
MOVE_PM        #60,@(R7+2*R6)        ;M[R7+*R6] = 60
                                     ;R7 = R7 + *R6
```



Register Indexed Addressing - Base Modified

Register Indexed Addressing – Base Modified

- Useful for stepping through data and for working with arrays.
- In ***C*** 2-D arrays are stored

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Register Indexed Addressing – Base Modified

- Useful for stepping through data and for working with arrays.
- In ***C*** 2-D arrays are stored

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

•In row major order [1,2,3;4,5,6;7,8,9]

Register Indexed Addressing – Base Modified

- Useful for stepping through data and for working with arrays.
- In ***C*** 2-D arrays are stored

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

•In column major format [1,4,7;2,5,8;3,6,9]

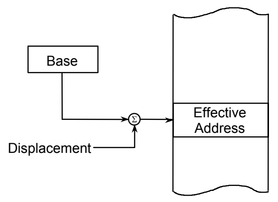
Register Indexed Addressing – Base Modified

- Hence for an M x N matrix the sum of the first column can be found as follows $\sum_{i=0}^{M-1} Array(i,0)$

```
  MOVE          #Array,R7          ;R7 = Array
  MOVE          #N,R6              ;R6 = N
  MOVE          #M-1,R5            ;R5 = M-1
  MOVE          #0,R0              ;R0 = 0
LOOP
  MOVE_PM       @(R7+R6),R1        ;R1 = M[R7+R6]
  ADD           R1,R0              ;R0 = R0 + R1
  SUBI          #1,R5              ;R5 = R5 - 1
  JGT           LOOP               ;Jmp if R5 > 0
```

Register Displaced Addressing

- With register displaced addressing the **effective address** is formed from the sum of a Base address (stored in a register) and a displacement contained in the instruction word.



Register Displaced Addressing

Register Indexed Addressing

- This addressing is particularly useful for accessing data structures.
- The start address of a particular structure is loaded into the Base register and the different elements of the structure are specified by the instruction, e.g.

```
struct      employee {
    long    emp_num;
    long    salary;
    long    age;
};
employee    Paul; employee Peter;
```

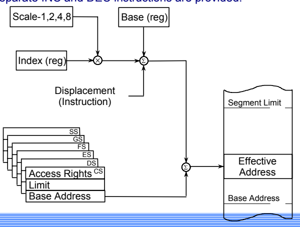
- The compiler will allocate 3 words to each structure Peter and Paul and will assign the start address of each structure to Paul & Peter; emp_num, salary & age will be given the values 0, 1 & 2.
  - Now can set the values of the structure using Base-Displacement addressing.

```
MOVE        #Peter,R7                 ;R7 = Peter
MOVE        #605,@(R7+emp_num)        ;M [Peter.emp_num] = 605
MOVE        #10000,@(R7+salary)       ;M [Peter.salary] = 10000
MOVE        #23,@(R7+age)             ;M [Peter.age] = 23
```

386 Addressing Modes

Intel 386 Address Generation Hardware

- All of the above addressing modes (except memory indirect) are supported by the 386 family of processors (486, Pentium are the same architecture as 386).
  - Base modification and automatic increment/decrement are not supported,
    - separate INC and DEC instructions are provided.



386 Addressing Modes

- There are **6** segment registers on the 386;
  - **CS**, **DS**, **SS**, **ES**, **FS**, **GS**, essentially code, data, stack and 3 extended segments.
- These segment registers point to a segment descriptor register which contains a **32-bit base address**, a **20 bit limit** and **12 bits** for access rights.
  - For each memory access the 32 bit base address is added to the relative address calculated from the instruction,
  - The address is checked to verify it is within the limit allow for that segment and the access rights are checked.
  - The segment register used depends on the instructions word,
  - CS is used for code access, DS for data access, SS for stack operations.

RISC vs. CISC

- RISC processors are usually 0-Address machines with registers
  - usually only support a small number of addressing modes.
  - All arithmetic and addressing operations act on internal registers.

RISC vs. CISC

- CISC processors are usually 1-Address machines with registers.
  - Instructions are usually 1-word with 2-word instructions for long constants (e.g. 32 bit constants).
  - Most instructions are multi-cycle, in 386 simple instruction are 2 to 8 cycles, others are > 20 cycles
- They allow arithmetic operations on memory,
  - e.g. add reg to memory which requires a memory read, add and memory write.
- They support a wide range of addressing modes.
- The main force driving/holding back CISC designs is the need to maintain Binary compatibility with previous members of the family,
  - e.g. Pentium will still run 8086 code, 68040 will still run 68000 code.

RISC vs. CISC

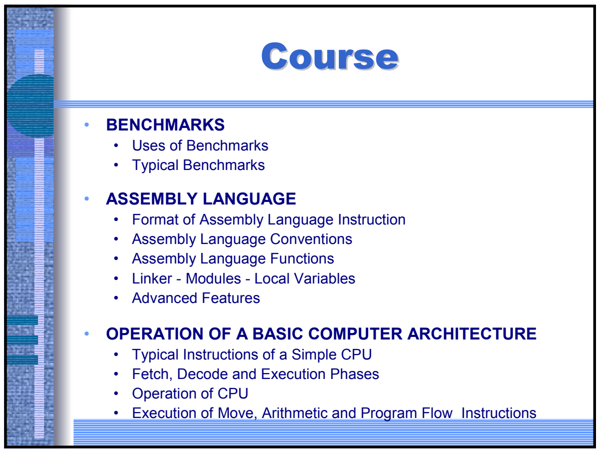
- **Accuracy of these architecture designs can be seen in the following example:**
- **CISC Motorola MC68000**
  - **contains 200,000 transistors,**
  - **10 person years of design effort**
  - **achieves a performance of 2 MIPS.**
- **RISC ARM Processor**
  - **contains 27,000 transistors,**
  - **6 person years of design effort**
  - **achieves a performance of 5 MIPS.**

RISC vs. CISC

- **RISC removes redundant instructions from the instruction set**

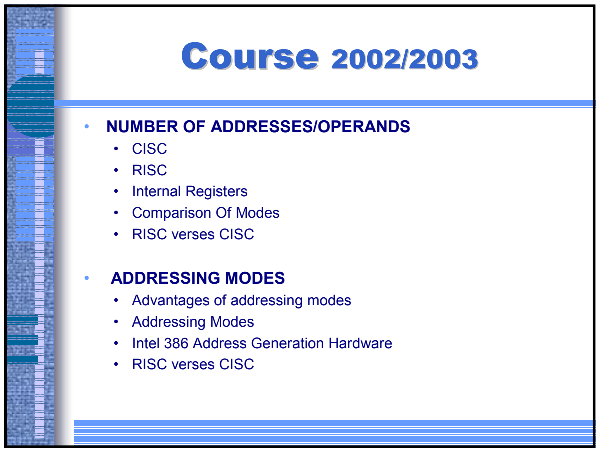
⇒Remaining instructions can be executed much more rapidly than was previously possible.

⇒Improved performance overall.



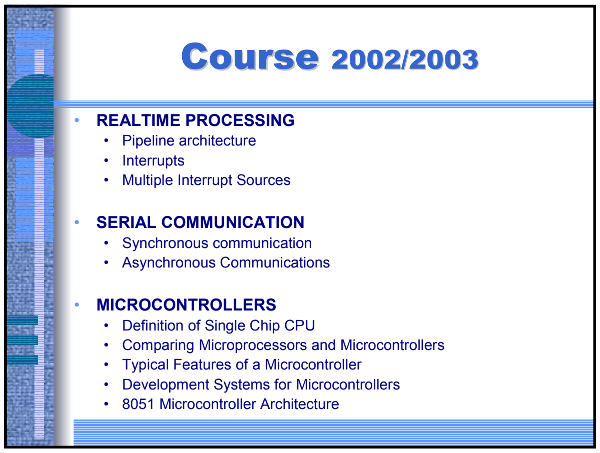
Course

- **BENCHMARKS**
  - Uses of Benchmarks
  - Typical Benchmarks
- **ASSEMBLY LANGUAGE**
  - Format of Assembly Language Instruction
  - Assembly Language Conventions
  - Assembly Language Functions
  - Linker - Modules - Local Variables
  - Advanced Features
- **OPERATION OF A BASIC COMPUTER ARCHITECTURE**
  - Typical Instructions of a Simple CPU
  - Fetch, Decode and Execution Phases
  - Operation of CPU
  - Execution of Move, Arithmetic and Program Flow Instructions



Course 2002/2003

- **NUMBER OF ADDRESSES/OPERANDS**
  - CISC
  - RISC
  - Internal Registers
  - Comparison Of Modes
  - RISC verses CISC
- **ADDRESSING MODES**
  - Advantages of addressing modes
  - Addressing Modes
  - Intel 386 Address Generation Hardware
  - RISC verses CISC



Course 2002/2003

- **REALTIME PROCESSING**
  - Pipeline architecture
  - Interrupts
  - Multiple Interrupt Sources
- **SERIAL COMMUNICATION**
  - Synchronous communication
  - Asynchronous Communications
- **MICROCONTROLLERS**
  - Definition of Single Chip CPU
  - Comparing Microprocessors and Microcontrollers
  - Typical Features of a Microcontroller
  - Development Systems for Microcontrollers
  - 8051 Microcontroller Architecture